Direct Application Launch from System Startup in Windows Vista and Windows 7

March 12, 2010

Abstract

The Windows Vista® and Windows® 7 operating systems provide built-in support for a fast system startup experience that boots or resumes directly into media or other applications. This support, called *direct application launch*, is possible on PCs that are running Windows Vista or Windows 7 by making simple changes to platform firmware and underlying platform wake circuitry. This paper describes the changes in platform hardware and firmware to support direct application launch that is based on button-press or wireless receiver events. It provides guidelines for system designers and firmware developers to implement platform support for direct application launch on PCs that run Windows Vista or Windows 7.

This information applies to most versions of the following operating systems:

Windows 7
Windows Vista

Note: Direct Application Launch does *not* run on Windows 7 Starter Edition or Windows Vista Starter Edition.

The current version of this paper is maintained on the Web at:

http://www.microsoft.com/whdc/system/vista/DirAppLaunch.mspx

References and resources discussed here are listed at the end of this paper.

***Disclaimer**: This document is provided "as-is". Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.*

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2010 Microsoft Corporation. All rights reserved.

Document History

Date	Change
March 12, 2010	Revised paper to indicate that Direct Application Launch works on both Windows Vista and Windows 7, but does *not* apply to Windows 7 Starter Edition or Windows Vista Starter Edition
November 18, 2005	First publication

Contents

Introduction 3
Advantages of Direct Application Launch in Windows Vista and Windows 7 3
Design Overview 5
  Functional Block Components 5
  General Design Approach and Event Flow 6
Implementation Details 6
  Platform Hardware Support for the Application-Launch Button 6
  Firmware Support 7
    Firmware Functional Responsibilities 7
  ACPI Support 7
    Button ACPI Declaration 7
    ACPI Method and Object Support 8
    Other Firmware Changes 9
    ACPI Driver Support 9
    Retrieving the Button Descriptor 10
  ACPI Handling of Button-Press Events 10
  Application-Launch Button Event Notifications to Platform Software 11
    User-Mode Software Notification Example 11
    Application-Launch Button-Event GUID 11
    Application-Launch Button Notification and Data Payload 11
  Button Agent and Application Launch 12
    Example Configuration 13
  Event Flows for Application-Launch Events 14
    Run-time Application Launch Event Data Flow 14
    Wake Button-Press Event Data Flow 15
    System Start from S5 Button-Press Event Data Flow 17
Next Steps 18
Resources 18

© 2010 Microsoft Corporation. All rights reserved.

Introduction

The Windows Vista® and Windows® 7 operating systems provide built-in support for launching applications directly from system startup. Direct application launch leverages existing Windows Vista and Windows 7 support for OnNow power management initiatives and technologies, including the platform sleep states and wake capabilities that are defined by the Advanced Configuration and Power Interface (ACPI) specification and supported by the Windows Driver Model (WDM), the underlying operating system, and platform hardware.

Manufacturers can take advantage of direct application launch in Windows Vista and Windows 7 to add consumer-friendly application-access buttons through chassis front panels or wireless remote controls to their system designs.

A common example of such a control is a media playback button. Typically, a media button is dedicated either to starting the system from an off state or to waking the system from a sleeping state and then entering a dedicated media playback mode. Normally, resuming from sleep returns the system to the state and context from which the operating system was suspended and booting the system from the off state presents the logon or user's desktop screen, depending on user account and password configurations. However, the desired experience for systems that feature media playback might be to start or wake the computer through a special-purpose button and immediately (as soon as the system is running) present the user with a media player or dedicated media shell.

This paper explores solutions that leverage the capabilities of a PC that is running Windows Vista or Windows 7 to implement a fast system-startup-to-application-launch experience through a single button press.

Advantages of Direct Application Launch in Windows Vista and Windows 7

Implementing consumer application controls and system states by using the built-in support for direct application launch from system startup provides numerous benefits to both the system builder and the end user. This implementation:

- Uses a single instance of both the Windows Vista or Windows 7 operating system and existing firmware:
  - No dual boot screens.
  - Ability to exit the media shell and return to the normal working environment without rebooting.
  - Windows desktop.
- Supports transitions from all system sleep states and soft-off states, including:
  - Standby (ACPI S3).
  - Hibernate (ACPI S4).
  - Shutdown (ACPI S5).
- Operates after the system is already started to enable direct application launch during run time.

© 2010 Microsoft Corporation. All rights reserved.

- Leverages investments in fast boot and resume initiatives to offer quick system startup:
  - Resume from standby (S3) ≤ 3 seconds.
  - Resume from hibernate (S4) ≤ 10 to 13 seconds.
  - Boot from off (S5) ≤ 18 to 25 seconds.
- Allows manufacturers to easily develop and innovate by supporting hardware buttons, wireless receivers, or both.
- Allows for rich Windows power management support at run time:
  - Optimal battery life on laptop designs.
  - Ability for manufacturers to leverage the Windows Vista and Windows 7 power management infrastructure for further innovation and platform support.
- Reduces complexity of manufacturer implementations:
  - Supported by industry standards.
  - No additional drivers or device support required for an additional operating system, an alternate set of codecs, or an additional disk partition.
  - Single operating system image to build, deploy, and support.
  - Simple metaphor: Press a button and get an experience (quickly).

Windows Vista and Windows 7 eliminate some of the disadvantages of design solutions that exist in the marketplace today. These solutions often require a separate firmware boot environment, operating system, or disk partition to facilitate quick system startup and to host the media shell, media applications, and digital media content and metadata. Implementing such solutions imposes additional burdens on manufacturers, including:

- No concurrent scenarios in alternate boot environment (for example, the user cannot read e-mail while listening to music).
- Software licensing overhead.
- Separate, dedicated mini-firmware code.
- An additional disk partition.
- Another set of device drivers, operating system, codecs, and media application to qualify and support.
- Duplicate disk imaging for each boot environment.
- Additional disk image size and factory build time and complexity.
- A fragmented, confusing user experience that requires the user to reboot to switch between the Windows shell and the dedicated application environment, which makes Windows experiences less available.

© 2010 Microsoft Corporation. All rights reserved.

Design Overview

This section provides high-level implementation details for the system-startup-to-direct-application-launch solution in Windows Vista and Windows 7.

Functional Block Components

Direct application launch in Windows Vista and Windows 7 consists of three fundamental functional blocks, as shown in Figure 1.

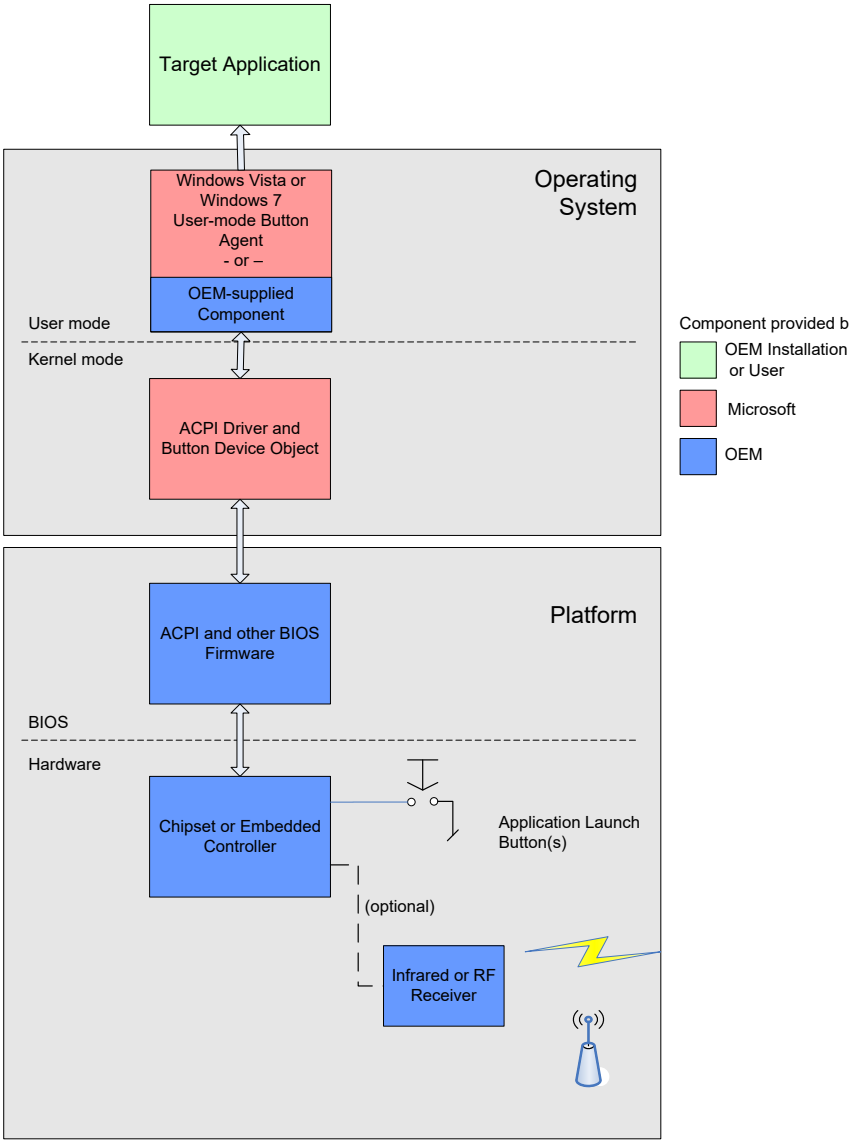

Figure 1. System Startup and Application Launch Functional Blocks

Starting from the bottom of Figure 1, these components consist of:

- One or more special-purpose application-launch buttons and associated wake circuitry in hardware.

  Optionally, a wireless infrared or radio frequency (RF) receiver could be used in place of or together with the application-launch button.

© 2010 Microsoft Corporation. All rights reserved.

- Platform firmware support to enable the button(s) or wireless receiver and preserve the system wake source (that is, which button was pressed). This includes ACPI and non-ACPI firmware support.
- Operating system components to retrieve the system wake source from firmware and launch the target application.

General Design Approach and Event Flow

Support for system startup and direct application launch is relatively simple:

- The platform provides one or more special-purpose buttons or wireless receivers (hereinafter collectively referred to as “buttons”) and the associated system-wake circuitry. Each button is described with a new Plug and Play hardware ID that the Windows ACPI driver recognizes and on which it loads the ACPI driver.
- Platform firmware enables the buttons to power on or to wake the system. Firmware includes the capability to detect and preserve the system wake source (that is, which button was pressed). This can include both ACPI and non-ACPI firmware support.
- The ACPI firmware for the button also returns a buffer that indicates the intended use of the button (such as a media button, Internet button, or calculator button). The manufacturer assigns the actual value that is used for this designation.
- When the system is booted, the Windows ACPI driver enumerates all instances of the new application-launch button, retrieves each button’s intended function value, and stores it in the registry.
- When the button is pressed, Windows Vista and Windows 7 use normal ACPI mechanisms to send a Notify code to the button’s device object. The Windows ACPI driver then triggers the Windows power manager to send a specialized power event to any registered listeners. The event data payload includes the unique button instance that was pressed.
- A user-mode button agent that is supplied by Windows Vista and Windows 7 receives this power management event (PME) notification and matches the button instance ID to the target application as described in the registry. The button agent then starts the target application.

Implementation Details

The example in this section describes a single button that is used to wake the system and launch a media player application. In practice, any number of buttons can be implemented and a button's key press can be associated with any application launch or control event (such as Mail, Media Shell, Play, or Pause).

Platform Hardware Support for the Application-Launch Button

Platform hardware:

- Handles run-time user button-press events (from the ACPI S0 state) by asserting the proper general-purpose event (GPE) and triggering the system control interrupt (SCI).
- Wakes the system from a sleep state (ACPI S1-S4).
- Optionally, starts the system from soft off (ACPI S5).

For a specific implementation, the manufacturer determines whether the button transitions from the off state or the sleep state.

© 2010 Microsoft Corporation. All rights reserved.

The application-launch button should normally be wired to any available GPE on an ACPI-compatible chip set. It could also be wired to a laptop's embedded controller (EC) or to a general-purpose I/O (GPIO) in the chipset that can be programmed to assert a GPE. The media button should be wired and the GPE should be programmed to function only as a wake event, and never as a run-time-only event.

Firmware Support

Platform firmware support consists of both ACPI and non-ACPI firmware.

Firmware Functional Responsibilities

The system firmware:

- Declares the button object(s) in ACPI.
- Provides the button's ACPI configuration and operational support.
- Correctly enables the special-purpose buttons, such as arming the button for wake from ACPI sleep states or from the S5 state.
- Captures and correctly identifies the system-wake or startup button press.
- Preserves the wake-source event across the firmware power-on self-test (POST) phase during transitions from the ACPI S4 or S5 states.
- Provides standard ACPI device configuration, method, and event support for the button device, such as GPE handlers, embedded-controller _Qxx event handlers, _STA and _HID methods, and so on.
- Provides the Microsoft-defined GHID method (shown in Figure 2 on the following page) for retrieving the button's intended function.

ACPI Support

Important

The sample implementation described in this paper and shown in Figure 2 depicts pseudocode that is intended only for illustrative purposes. This sample is incomplete and is not suitable for inclusion in production ACPI source language (ASL).

Button ACPI Declaration

A device that represents the media button is placed in the _SB scope of the ACPI namespace, as shown in Figure 2. The button is described with a Plug and Play hardware ID of PNP0C32. The operating system ejects a physical device object (PDO) device node for this device, associates this device object with the Windows *acpi.sys* driver, and loads the driver on this device as the functional device object (FDO).

© 2010 Microsoft Corporation. All rights reserved.

Figure 2 shows a sample ACPI namespace for a direct application-launch button.

```
Device(\_SB_PCI.LPC.EC_MBTN) {    // media button wired on embedded
controller
    Name(_HID, PNP0C32)     // HIDACPI button
    Name(_UID, 1)           // unique instance ID
    Method(_STA, 0x0, NotSerialized) {
        Return(0x0F)        // optional – do OEM-specific actions here
    }
    Name(_PRW, Package(2) {
        1,              // bit 1 of GPE to enable system startup
        0x04}           // can wake up from S4 state
        )
    }
    Method (GHID, 0x0) {    // returns descriptor of button instance
        If (<BTNW>) {       // platform-specific wake source detection
            Notify(\_SB_PCI.LPC.EC.MBTN, 0x02)
        }
    Return(Buffer(){0x01}) // UsageID of button; maps to app to launch
}
…
Scope(\_GPE){           // Root-level event handlers
    Method(_L01) {      // uses bit 1 of GP0_STS register
        If (<BTNW>) {    // platform-specific wake source detection
            Notify(\_SB_PCI.LPC.EC.MBTN, 0x02)
        }
        If (<BTNP>) {    // platform-specific run-time press detection
            Notify(\_SB_PCI.LPC.EC.MBTN, 0x80)
        }
    }                   // end of _L01 handler
}                       // end of \_GPE scope

…
Method(_WAK, 0x1, NotSerialized) {
        If (<BTNW>) {    // platform-specific wake source detection
            Notify(\_SB_PCI.LPC.EC.MBTN, 0x02)
        }
}
```

Figure 2. Example ACPI Namespace

ACPI Method and Object Support

The following ACPI objects and methods are placed under the scope of the button device in the ACPI namespace, as shown in Figure 2.

- **_HID**: This method is required. _HID must be described with the specific object value for the HIDACPI button device Plug and Play hardware ID of *PNP0C32*.
- **_UID:** This method is optional. _UID returns the appropriate unique ID for the media button device. _UID is required only if multiple instances of the same device occur in the namespace. Note that _UID must evaluate to a decimal numeric value.
- **_STA:** This method is optional. Note that after the operating system has detected the presence of the direct application launch button and the HotStart service has been configured to launch an application based on button press events for this device, reporting that the button device is not present via _STA may not be used to disable that event from triggering the associated application launch.
- **_PRW:** This method is required. _PRW returns the GPE pin and system wake level for the button device. In most cases, the system wake level should be “S4”.

© 2010 Microsoft Corporation. All rights reserved.

- **_PSW:** This method is optional. The operating system runs this method to set the device-specific registers to enable the wake circuitry for the button device.
- **Event handling:** A _GPE._Lxx, _GPE_Exx, or _Qxx event handler for the device is required. <xx> corresponds to the GPE pin to which the button is connected and is described in the _PRW package for the device. The event handler must issue the following Notify codes to the application-launch button device object:
  - Notify(<*btn*>, 0x80) whenever the button is pushed at run time.
  - Notify(<*btn*>, 0x02) when firmware determines that the button-press event was the system-wake event.
- **GHID method:** This method is required. The Microsoft-defined GHID method indicates the desired function, or "role," of the button, such as media, Internet, or calculator. This description is used to help associate the button with the correct application to launch. The GHID method returns a buffer that contains a value that indicates the button function. The manufacturer chooses the actual value, which can be mapped to the target application by using the Windows registry. The GHID method can return a BYTE, WORD, or DWORD. The value must be encoded in little-endian byte order (least significant byte first). Implementation of the GHID method is shown in Figure 2 earlier in this paper. The buffer returned by GHID must be in the form of a decimal number.

Other Firmware Changes

To enable the operating system to determine the source of the S0 transition, hardware or firmware must detect and save the source of the wake event so that it can be returned later during evaluation of the manufacturer-specific ASL implementation, as illustrated by the *BTNW* placeholder method shown in Figure 2 earlier in this paper. If firmware fails to save the wake source, it cannot properly issue the Notify codes that are described in this paper, so the system would wake but the target application would not be launched. Care must be taken to ensure subsequent button-press events do not overwrite the original wake source.

ACPI Driver Support

The ACPI driver *acpi.sys*:

- Retrieves the HID description for each unique button device that is described in the ACPI namespace and makes this description available to the supplied user-mode button agent or other user-mode components.
- Handles both run-time and wake button-press Notify events that are issued from the platform's ACPI namespace.
- Interfaces with the kernel power manager to issue power notifications of button-press events.

Button Device Objects

The direct application-launch button device has the following Plug and Play hardware ID:

PNP0C32

The *acpi.sys* driver object creates one device object for each unique application-launch button device that is declared in the ACPI namespace.

© 2010 Microsoft Corporation. All rights reserved.

Button Driver Interface Device Class

The device class for a direct application-launch button is as follows:

```
Application Launch Buttons
Class = AppLaunchButtonClass
ClassGuid 4D36E97D-E325-11CE-BFC1-08002BE10318
```

Retrieving the Button Descriptor

To properly identify the special-purpose button's function (for example, a Media button or an Internet button), firmware includes a value under each unique button instance in the ACPI namespace to act as a descriptor. The manufacturer determines the specific value, which can be configured through the Windows registry to launch any target application.

When the ACPI driver initializes, it reads this value by evaluating the GHID method, a Microsoft-defined ACPI method that returns a buffer that contains one or more DWORDs that describe the button's purpose.

The ACPI driver stores the contents of this buffer in the registry in the UserHidBlock value under the device registry key. This is the key that is returned by **IoOpenDeviceRegistryKey**(*deviceObject*), where <n> is the button's unique instance ID:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\ACPI\PNP0C32\<n>
\Device Parameters

Name: UserHIDBlock
Type: REG_BINARY
Data: <UsageID>
```

ACPI Handling of Button-Press Events

Run-time button-press or system-wake events are conveyed from platform firmware by using ACPI Notify codes. As part of the platform's typical GPE or EC event handling, the ACPI driver queues methods that correspond to the GPE index or EC query code to run. Firmware ASL issues Notify events to the button driver from these _Lxx or _Qxx methods. The Notify codes to be used are:

- Notify(*btn,* 0x80). A run-time button press occurred.
- Notify(*btn,* 0x02). A system-wake or power-on button press occurred.

When the Windows Vista or Windows 7 ACPI driver receives these events, it triggers a corresponding power manager event from the Windows power manager. User-mode software can then receive this event and launch the desired application.

Firmware must detect that the application-launch button started the system or caused it to wake, and it must store the wake source until ACPI has initialized and can consume that event. This may entail firmware boot code outside ACPI storing one or more variables that ASL can later examine. An ASL method to examine such a variable is suggested by the *<BTNW>* and *<BTNP>* placeholder methods shown in the sample ASL in Figure 2 earlier in this paper. The system firmware developer determines the exact implementation.

© 2010 Microsoft Corporation. All rights reserved.

Application-Launch Button Event Notifications to Platform Software

When the ACPI driver handles a Notify(*btn,* 0x80) or Notify(*btn,* 0x02) event, it calls into the Windows kernel power manager to send a power-setting notification event that is specific to the application-launch button press to any component that has registered to receive these notifications.

This notification is available to kernel-mode device drivers or user-mode services and applications. Any system software component can detect and act on these application-launch button-press events, which provides maximum flexibility and extensibility for platform development.

User-Mode Software Notification Example

To listen for application-launch button-press events, system software components can register for notifications by using the standard Windows Vista and Windows 7 power function, **RegisterPowerSettingNotification**.

```
HANDLE RegisterPowerSettingNotification(
       IN HANDLE hRecipient,
       IN CONST LPGUID PowerSettingGuid,
       IN DWORD Flags);
```

In this function:

- The *hRecipient* parameter is a handle to the window or service that is requesting the button-event notification.
- The *PowerSettingGuid* parameter is the GUID for the application-launch button-event notification, defined as GUID_APPLAUNCH_BUTTON.
- The *Flags* parameter can be either DEVICE_NOTIFY_WINDOW_HANDLE or DEVICE_NOTIFY_SERVICE_HANDLE to indicate whether the provided handle is for a window or a service.

For more information about power management functions in Windows Vista and Windows 7, see "Resources" at the end of this paper.

Application-Launch Button-Event GUID

The following GUID is the unique ID for registering for application-launch button-press events:

```
GUID_APPLAUNCH_BUTTON
1A689231-7399-4E9A-8F99-B71F999DB3FA
```

Application-Launch Button Notification and Data Payload

A window is notified of an application-launch button-press event by a **WM_POWERBROADCAST** message with a *wParam* of **PBT_POWERSETTINGCHANGE**. The *lParam* for this message is a pointer to the following structure:

```
typedef struct {
       GUID PowerSetting;
       DWORD DataLength;
       UCHAR Data[1];
} POWER_SETTING_BROADCAST, *PPOWER_SETTING_BROADCAST;
```


© 2010 Microsoft Corporation. All rights reserved.

The **Data** member of this structure points to a data payload that is delivered with each notification of the application-launch button-press event. This data payload has the following format:

```
typedef struct _APPLICATIONLAUNCH_SETTING_VALUE {

    //
    // System time when the most recent button press occurred.
    // Note that this is specified in 100ns internvals
    // since January 1, 1601.
    //
    LARGE_INTEGER       ActivationTime;

    //
    // Reserved for internal use.
    //
    ULONG               Flags;

    //
    // which instance of this device was pressed?
    //
    ULONG               ButtonInstanceID;

} APPLICATIONLAUNCH_SETTING_VALUE, *PAPPLICATIONLAUNCH_SETTING_VALUE;
```

Button Agent and Application Launch

Windows Vista and Windows 7 provide a user-mode button agent that subscribes to application-launch notifications from the Windows kernel power manager and launches the application that is associated with a specific button Usage ID in the Windows registry. This behavior is implemented as a task named HotStart that is launched by Windows Task Manager when the user logs on. The HotStart task can be viewed and configured in the Scheduled Tasks viewer management console in the Performance and Maintenance Control Panel application.

Configuring the Target Application

The manufacturer can configure the application that is associated with a specific Usage ID by using the following registry key:

```
HKLM\System\CurrentControlSet\Control\MobilePC\HotStartButtons\<UsageID>
```

<UsageID> corresponds to the value that is specified in the system's ACPI namespace. The GHID method returns this value for each application-launch button instance. Note that this registry key is not present in the registry by default; it must be added by the manufacturer when the system is configured to support direct application launch.

The value for this key should contain the full path to the application to be launched, including any command line parameters, as in the following example:

```
"C:\\Windows\\System32\\calc.exe"
```

Windows Vista and Windows 7 do not populate the application-launch button by default, and so make no associations between Usage ID values and target applications. The manufacturer can choose any value for Usage ID and map it to any target application.

© 2010 Microsoft Corporation. All rights reserved.

Example Configuration

The following example demonstrates how to configure a system that implements a direct application launch button to start an application. In this example:

- The BIOS returns the value 0x04 for the usage ID when the GHID method is evaluated, as shown in Figure 2 earlier in this paper.
- Windows Media® Player is the application that is launched when the application launch button is pressed.

To configure the HotStart task to launch Windows Media Player

1. Start the Registry Editor and select the following key:

   `HKLM\System\CurrentControlSet\Control`

2. Add the following key:

   `MobilePC`

3. Under the MobilePC key, add the following key:

   `HotStartButtons`

4. Under the HotStartButtons key, add a key whose name matches the UsageID returned by the BIOS GHID method for the button you are configuring. In this example, the BIOS returns 0x04 for the UsageID.

   `4`

5. Under the UsageID key added in step 4, add a new string value that is named ApplicationPath.
6. Edit the default value for this key and set this value to the full path of the application to launch:

   `C:\Program Files\windows media player\wmplayer.exe`

Figure 3 shows an example of the new key in the Registry Editor window.

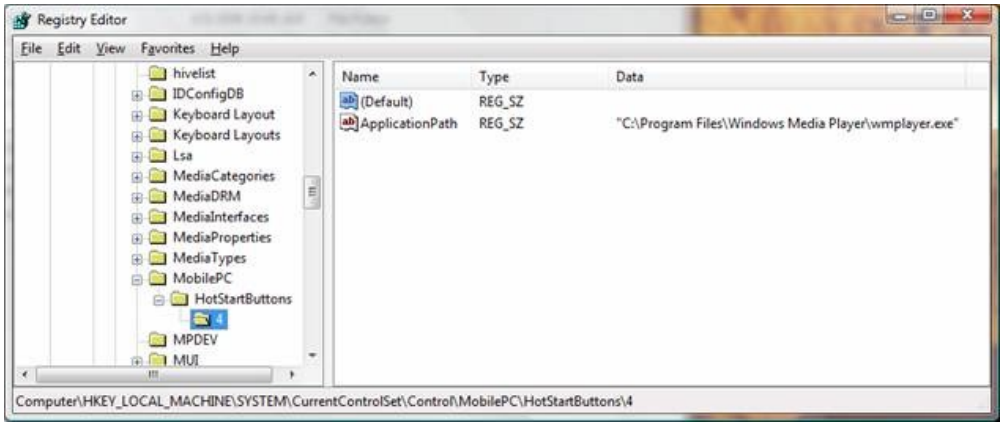

Figure 3. Example Configuration for Direct Application Launch

Preventing Startup of the HotStart Task

The HotStart task can be disabled through Group Policy by setting the following registry keys:

```
<HKCU/HKLM>\Software\Microsoft\Windows\CurrentVersion\Policies\System
"NoHotStart"=dword:0
```


© 2010 Microsoft Corporation. All rights reserved.

Event Flows for Application-Launch Events

The platform infrastructure that is described in this paper can be illustrated by tracing the event flows through each part of the system.

Run-time Application Launch Event Data Flow

Figure 4 shows the steps that are taken when the application-launch event is initiated at run time; that is, when the system is already in the S0 working state.

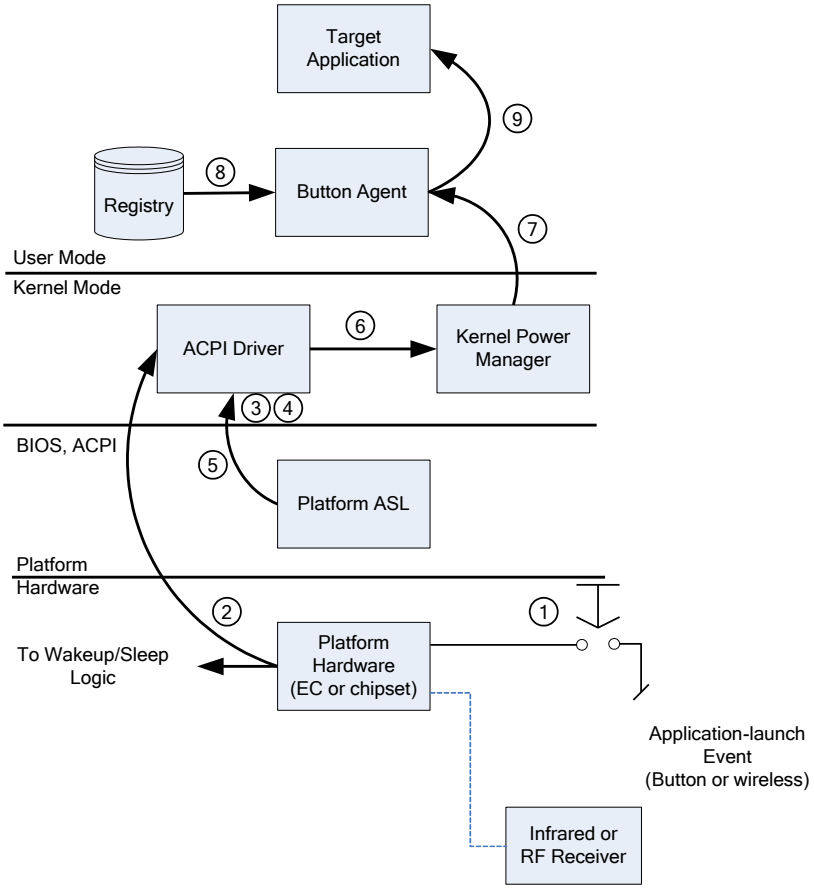

Figure 4. Run-time Application Launch Event

Starting at the bottom of Figure 4:

1. With the system in the working (S0) state, the user presses the application-launch button or activates the wireless remote control.
2. Platform hardware detects the button switch closure or wireless receiver event and asserts the appropriate GPE, which in turn causes the SCI to be asserted.
3. The ACPI driver runs the SCI code and determines that GPE*n* has been asserted.
4. The ACPI interpreter runs the _L0*n* method as part of its normal ACPI interrupt handling.
5. The ASL code for the _L0*n* method issues a Notify(*btn*, 0x80) event to the button device to indicate that the button was pressed at run time.

© 2010 Microsoft Corporation. All rights reserved.

6. The ACPI driver calls a private interface into the kernel power manager to invoke a power notification for the button-press event.
7. The kernel power manager sends an application-launch event to all registered listeners.
8. The user-mode button agent receives the application-launch event, including the unique instance ID of the button that was pressed. The button agent looks up the application-launch string that is associated with this button's instance ID.
9. The button agent then launches the target application.

Wake Button-Press Event Data Flow

Figure 5 shows the steps that are taken when the application-launch button is pressed to start the system from the ACPI S1-S4 sleeping states.

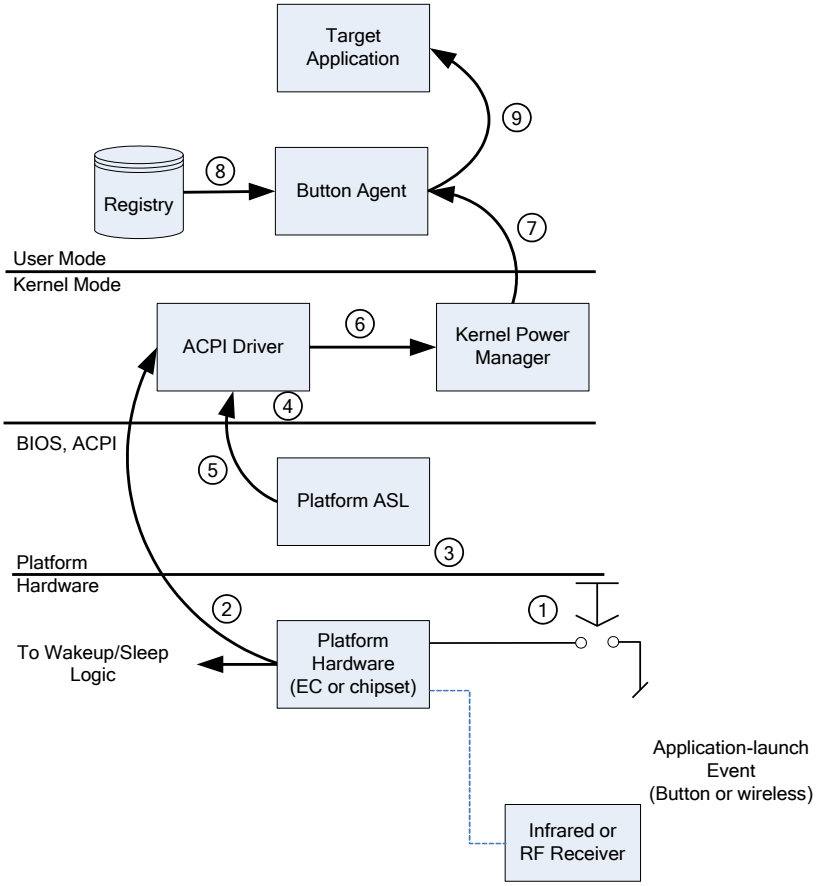

Figure 5. System Wake Button-Press Event

Starting at the bottom of Figure 5:

1. The system is in an ACPI system sleep state (S1-S4). The user presses the application-launch button or activates the wireless remote control.
2. Normal hardware mechanisms wake the system and return it to the S0 operating state.
3. System firmware preserves the system wake source (the button that was pressed).

© 2010 Microsoft Corporation. All rights reserved.

4. When the operating system resumes execution, the ACPI driver evaluates the _WAK method.
5. As part of the _WAK method, ASL runs a private method to determine if the application-launch button was the system-wake source. If so, it issues a Notify(*btn*, 0x02) event to the application-launch button object.
6. The ACPI driver calls a private interface into the kernel power manager to invoke a power notification for the button-press event.
7. The kernel power manager sends an application-launch event to all registered listeners.
8. The user-mode button agent receives the application-launch event, including the unique instance ID of the button that was pressed. The button agent looks up the application-launch string that is associated with this button's instance ID.
9. The button agent then launches the appropriate application.

© 2010 Microsoft Corporation. All rights reserved.

System Start from S5 Button-Press Event Data Flow

Figure 6 shows the steps that are taken when the application-launch button is pressed to start the system from the ACPI S5 soft-off state.

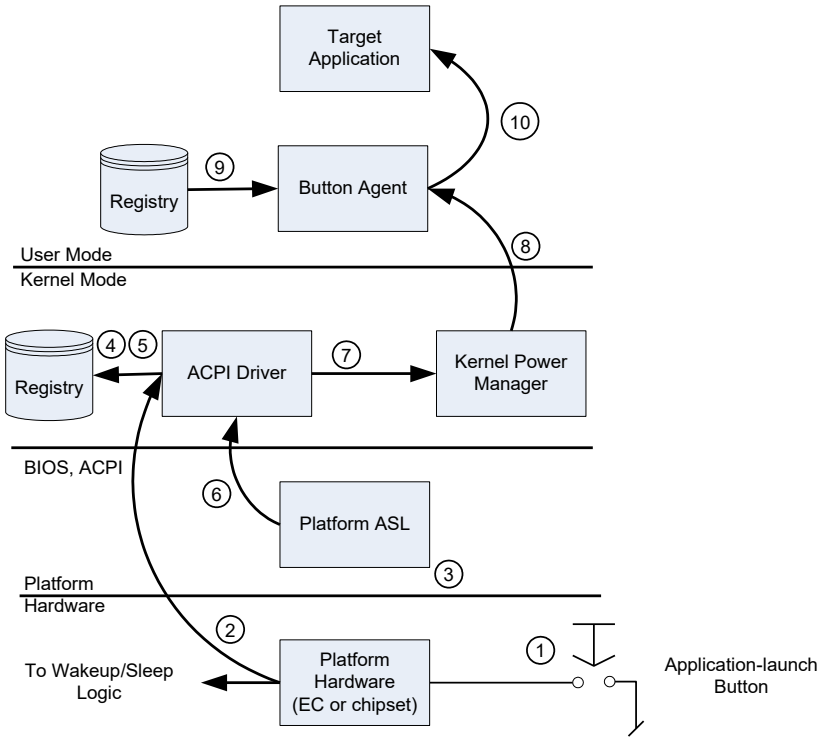

Figure 6. System Power-On Button-Press Event

Starting at the bottom of Figure 6:

1. The system is in the ACPI soft off state (S5). The user presses the application-launch button.
2. Normal hardware mechanisms start the system and return it to the S0 operating state.
3. System firmware preserves the system startup source (the button that was pressed).
4. As the ACPI driver initializes, it enumerates each instance of the PNP0C32 device in the ACPI namespace. For each device that is found, ACPI adds an instance ID to the Windows registry.
5. The ACPI driver evaluates the GHID method for each PNP0C32 object in the namespace and stores the Usage ID in the registry.
6. In the GHID method, firmware ASL evaluates the <*BTNW*> proprietary method to determine if this specific application-launch button instance started the system. If so, firmware ASL issues a Notify(*btn*, 0x02) event for that device instance.
7. The ACPI driver calls a private interface into the kernel power manager to invoke a power notification for the button-press event.

© 2010 Microsoft Corporation. All rights reserved.

8. The kernel power manager sends an application-launch event to all registered listeners.
9. The user-mode button agent receives the application-launch event, including the unique instance ID of the button that was pressed. The button agent looks up the application-launch string that is associated with this button's instance ID.
10. The button agent then launches the appropriate application.

Next Steps

The direct application-launch support in Windows Vista and Windows 7, together with industry-standard ACPI platform hardware and firmware support, provides a simple, flexible, and powerful development framework that can be easily leveraged to realize system-wake and application-launch scenarios, without incurring the additional expense and complication of alternate operating system or firmware components.

By using this infrastructure, manufacturers and system designers can use simple and reliable mechanisms to provide value-add hardware features to their product offerings.

Manufacturers are encouraged to:

- Consider enabling hardware buttons or wireless receivers in their platform designs to wake and launch media or other consumer applications.
- Leverage the direct application-launch button support in Windows Vista and Windows 7 to simplify and extend their application-launch button scenarios.

Resources

ACPI Specification – Revision 3.0
http://www.acpi.info/

ACPI / Power Management – Architecture and Driver Support
http://www.microsoft.com/whdc/system/pnppwr/powermgmt/default.mspx

Windows SDK
http://msdn.microsoft.com/en-us/windows/bb980924.aspx

Windows Driver Kit
http://msdn.microsoft.com/en-us/library/ms794193.aspx

© 2010 Microsoft Corporation. All rights reserved.